Summer Semester 2015

Software Engineering Design & Construction

Dr. Michael Eichberg
Fachgebiet Softwaretechnik
Technische Universität Darmstadt

Proxy Pattern

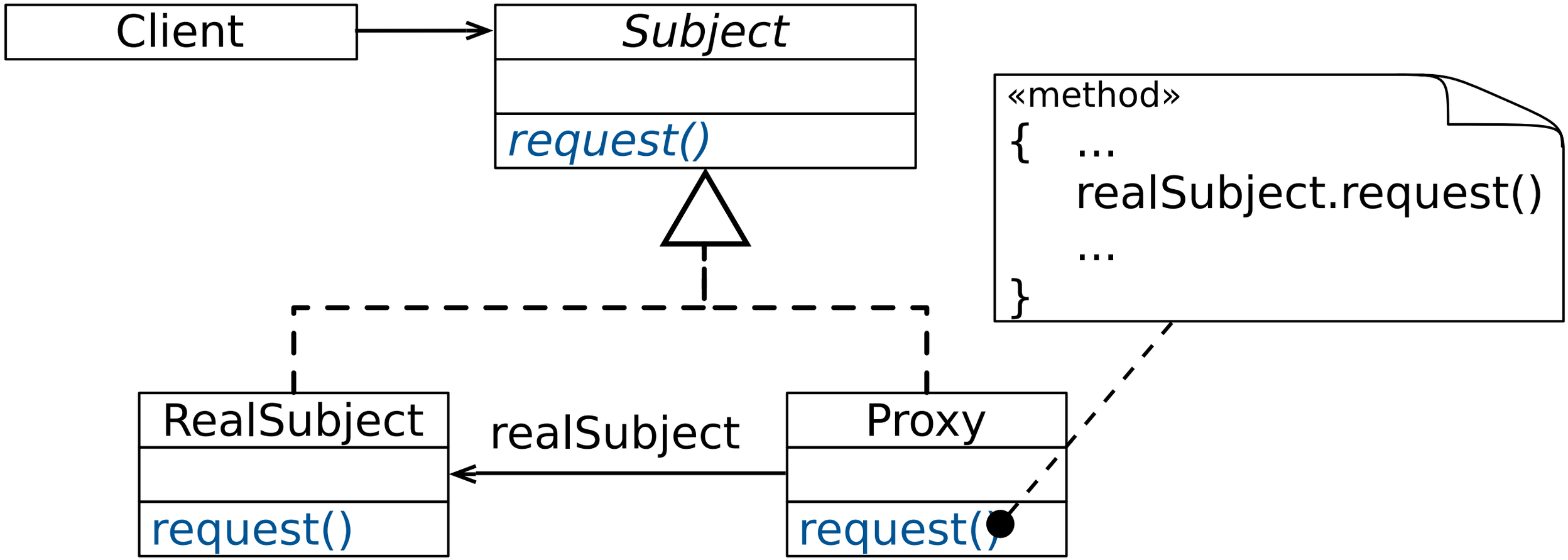
Proxy Pattern

Provide a surrogate or placeholder for another object to control access to it.

Proxy Pattern - Typical Variations

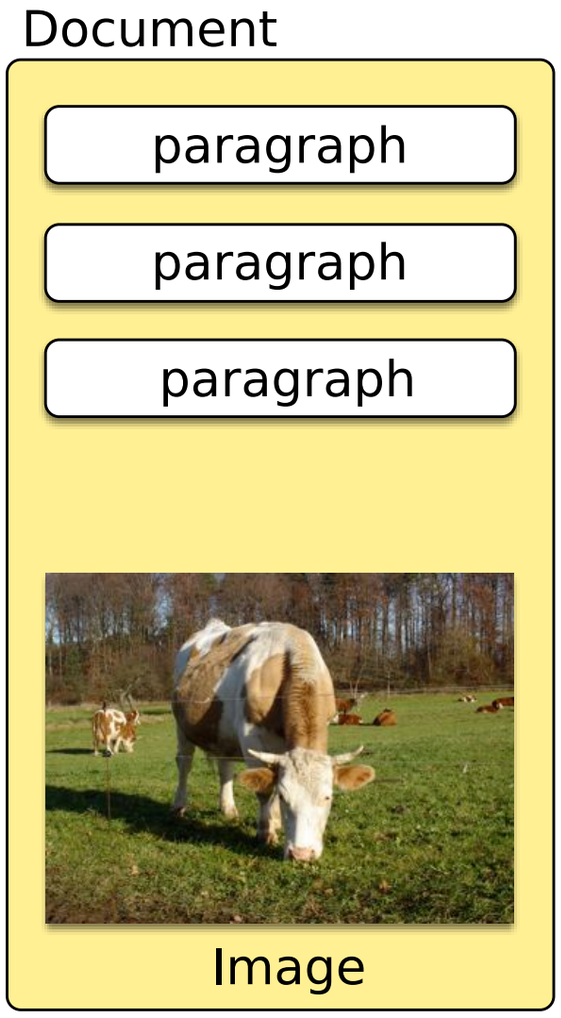
- Virtual Proxies: Placeholders
- Smart References: Additional functionality
- Remote Proxies: Make distribution transparent
- Protection Proxies: Rights management

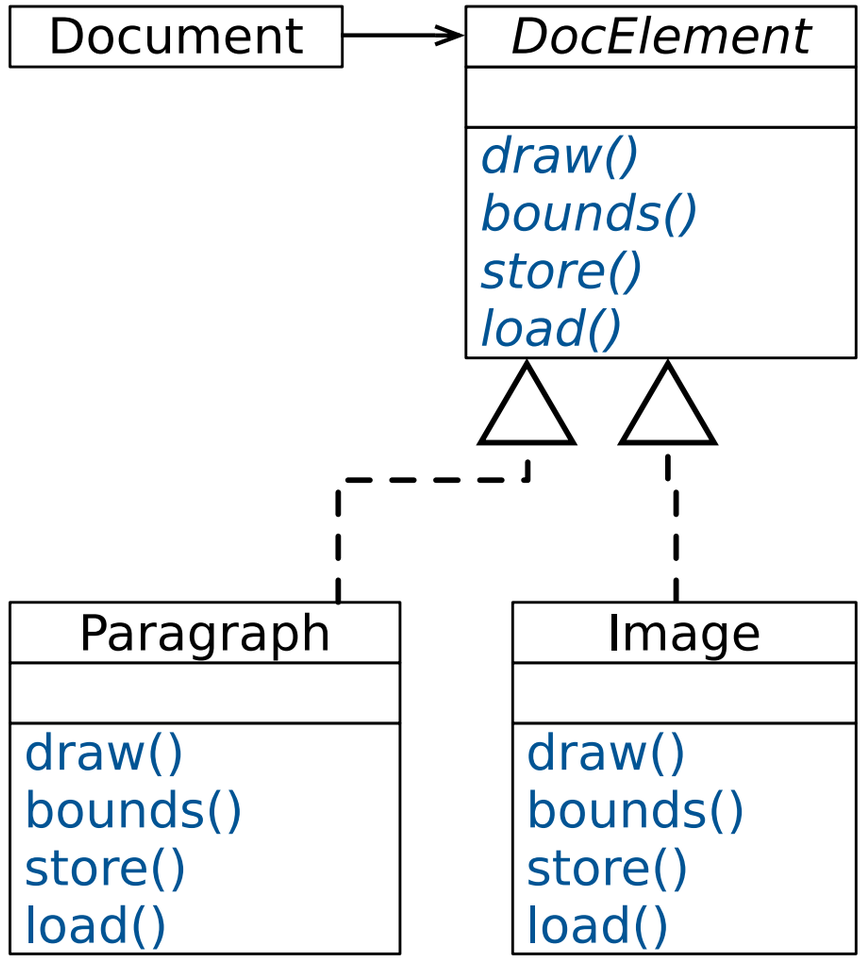
Proxy Pattern Structure



Example

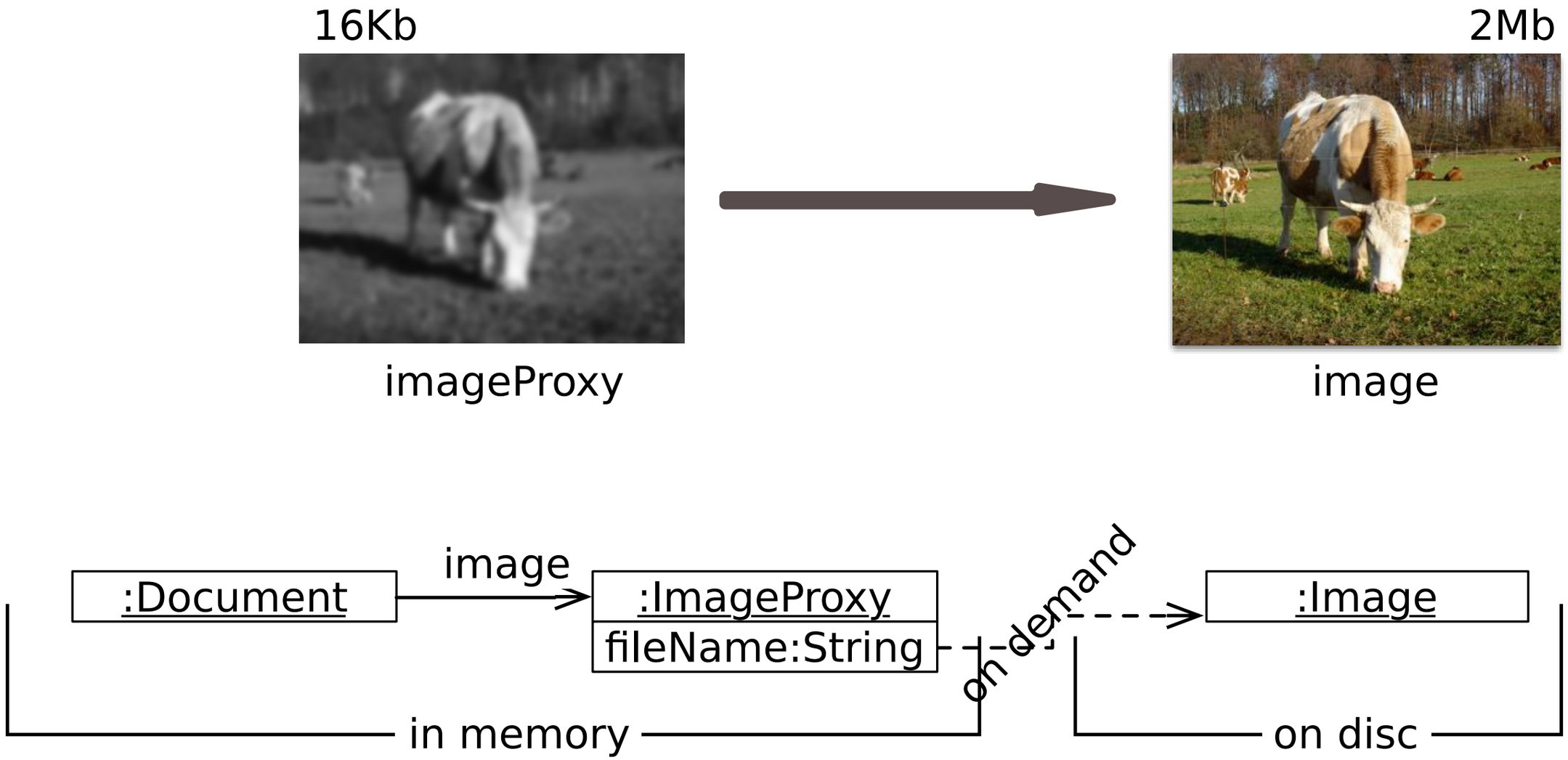
- Imagine, you are developing a browser rendering engine.
- In this case you do not want to handle all elements in a straightforward manner.
- E.g., you immediately want to start laying out the page even if not all images are already completely loaded. However, this should be completely transparent to the layout engine.





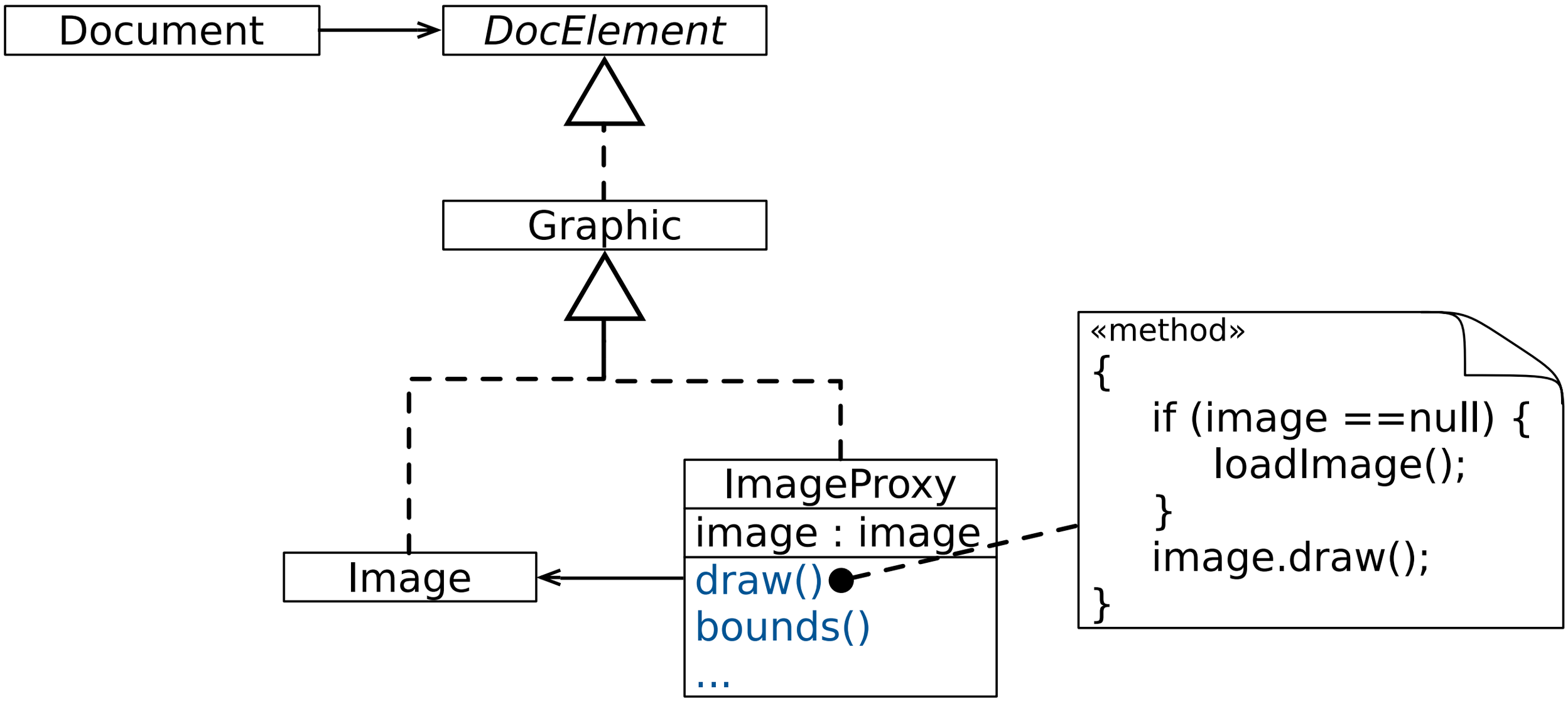
How can I hide the fact that loading the image takes time?

Lazy Loading - Solution



- We use another object, an image proxy, that acts as a stand-in for the real image.

Lazy Loading - Solution



Summary

The Proxy Pattern describes how to replace an object with a surrogate object.

- **without making clients aware of that fact,**
  (I.e., the client is not creating the proxy object and is usually has no direct dependency on the proxy's type.)

- while achieving a benefit of some kind:
  - lazy creation,
  - resource and/or rights management, or
  - distribution transparency.

Java's Dynamic Proxy Class

- A **dynamic proxy class** is a class that implements a list of interfaces specified at runtime such that a method invocation through one of the interfaces on an instance of the class will be encoded and dispatched to another object through a uniform interface.

- A **proxy interface** is such an interface that is implemented by a proxy class.

- A **proxy instance** is an instance of a proxy class.

Subtitle Text

Java's Dynamic Proxy Class - Example

```
public interface Foo { Object bar(Object obj); }
public class FooImpl implements Foo { Object bar(Object obj) { … } }

public class DebugProxy implements java.lang.reflect.InvocationHandler {
  private Object obj;

  public static Object newInstance(Object obj) {
    return Proxy.newProxyInstance(
      obj.getClass().getClassLoader(),obj.getClass().getInterfaces(),
      new DebugProxy(obj));
   }

  private DebugProxy(Object obj) { this.obj = obj; }

  public Object invoke(Object proxy, Method m, Object[] args) throws Throwable {
    System.out.println("before method " + m.getName());
    return m.invoke(obj, args);
  }
}
```

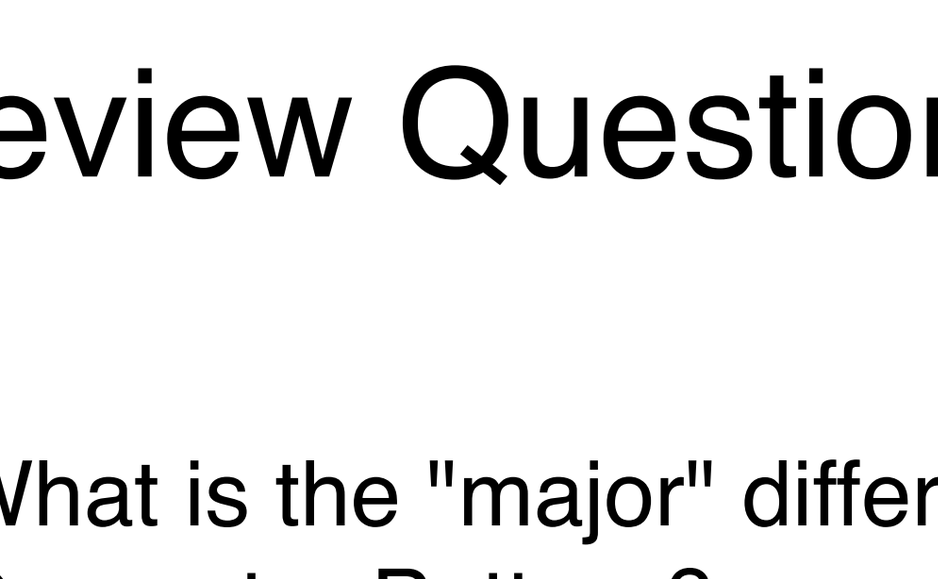
Setup

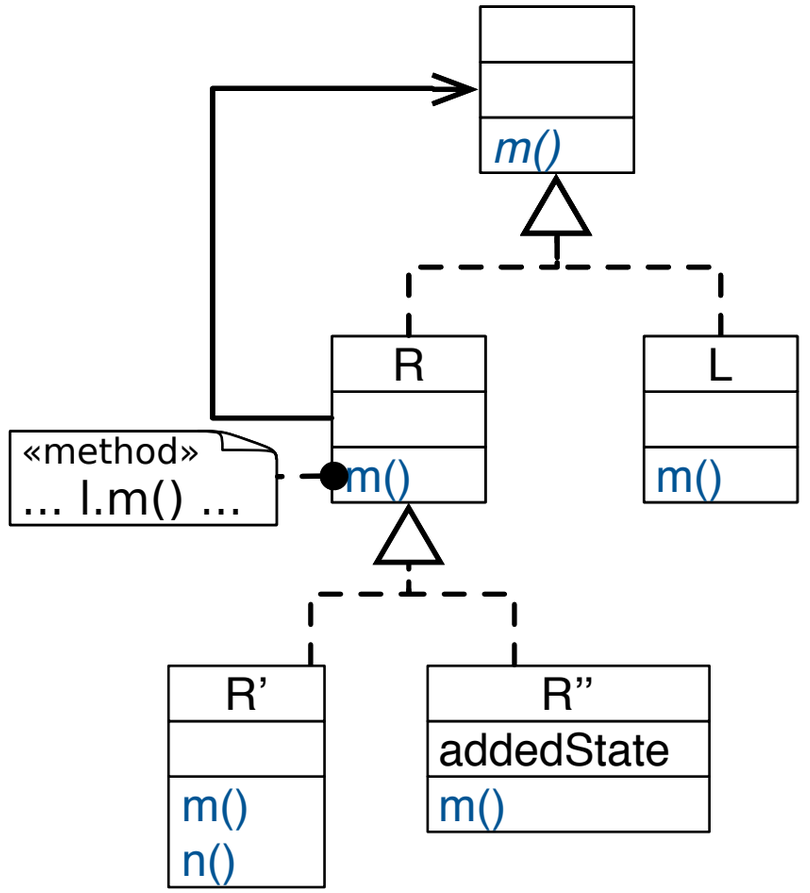
Usage

```
Foo foo = (Foo) DebugProxy.newInstance(new FooImpl());
foo.bar(null);
```

Review Questions

- What is the "major" difference between the Proxy and the Decorator Pattern?
  (Think about the structure and the behavior.)





Review Questions

- Is the Proxy Design Pattern subject to the "fragile base class" problem?
  (And if so, where and in which way?)

- In Java, we only have forwarding semantics, but **could it** be desirable to have delegation semantics, when implementing the proxy pattern?